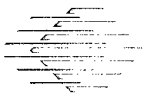


Carnegie Mellon
Software Engineering Institute

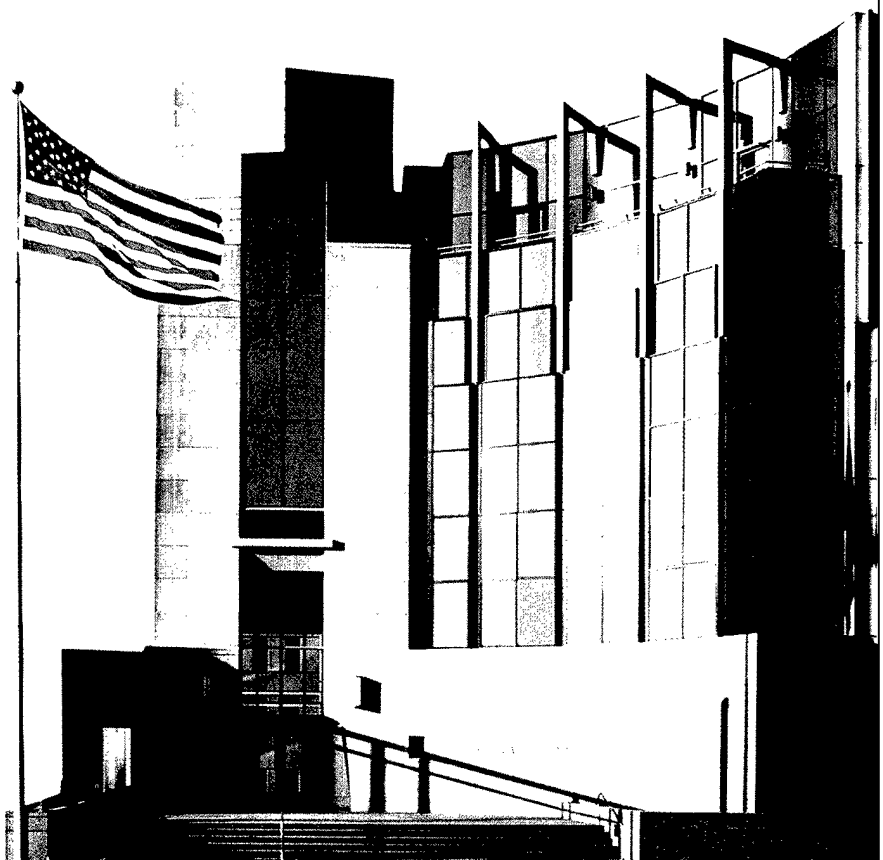
Quality Attribute Design Primitives

Len Bass
Mark Klein
Felix Bachmann

December 2000

Architecture Tradeoff Analysis Initiative

Technical Note
CMU/SEI-2000-TN-017



Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admission, employment, or administration of its programs or activities on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state, or local laws or executive orders.

In addition, Carnegie Mellon University does not discriminate in admission, employment or administration of its programs on the basis of religion, creed, ancestry, belief, age, veteran status, sexual orientation or in violation of federal, state, or local laws or executive orders. However, in the judgment of the Carnegie Mellon Human Relations Commission, the Department of Defense policy of "Don't ask, don't tell, don't pursue" excludes openly gay, lesbian and bisexual students from receiving ROTC scholarships or serving in the military. Nevertheless, all ROTC classes at Carnegie Mellon University are available to all students.

Inquiries concerning application of these statements should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-6684 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-2056.

Obtain general information about Carnegie Mellon University by calling (412) 268-2000.

Quality Attribute Design Primitives

Len Bass
Mark Klein
Felix Bachmann

December 2000

Architecture Tradeoff Analysis Initiative

Technical Note
CMU/SEI-2000-TN-017

Contents

List of Figures

List of Tables

Acknowledgements

We have discussed these ideas with a large number of our colleagues. We would like to thank Bob Ellison, Peter Feiler, Bonnie E. John, Rick Kazman, Judy Stafford, Nick Graham, John Lehoczky, Andy Moore, Robert Nord, Linda Northrop, Greg Phillips, and Bill Wood for their contributions to the evolution of these ideas.

Preface

This technical note presents a snapshot of work in progress. In making it available, we are looking to the software community for support. Your reaction and input will help us advance the concepts in this paper. In the future, we will publish similar snapshots as this work evolves.

Abstract

This report focuses on the quality attribute aspects of architectural mechanisms. An architectural mechanism is "a structure whereby objects collaborate to provide some behavior that satisfies a requirement of the problem" [Booch 96]. This report addresses mechanisms that significantly affect quality attribute behavior and have sufficient content for analysis. Codifying such mechanisms will enable architects to identify the choices necessary to achieve quality attribute goals. This, in turn, will set a foundation for further software architectural design and analysis.

1 Introduction

Over the last 30 years, a number of researchers and practitioners have examined how systems achieve software quality attributes. Boehm and the International Organization for Standardization (ISO) introduced taxonomies of quality attributes [Boehm 78, ISO 91]. Bass, Bergey, Klein, and others have analyzed the relationship between software architecture and quality attributes [Bass 98, Bergey 01, Klein 99a, Klein 99b]. Various attribute communities have mathematically explored the meaning of their particular attributes [Kleinrock 76, Lyu 96]. The patterns community has codified recurring and useful patterns in practice [Gamma 95]. However, no one has systematically and completely documented the relationship between software architecture and quality attributes. The lack of documentation stems from several causes:

- The lack of a precise definition for many quality attributes inhibits the exploration process. Attributes such as reliability, availability, and performance have been studied for years and have generally accepted definitions. Other attributes, such as modifiability, security, and usability, do not have generally accepted definitions.
- Attributes are not discrete or isolated. For example, availability is an attribute in its own right. However, it is also a subset of security (because denial of service attack could limit availability) and usability (because users require maximum uptime). Is portability a subset of modifiability or an attribute in its own right? Both relationships exist in quality attribute taxonomies.
- Attribute analysis does not lend itself to standardization. There are hundreds of patterns at different levels of granularity, with different relationships among them. As a result, it is difficult to decide which situations or patterns to analyze for what quality, much less to categorize and store that information for reuse.
- Analysis techniques are specific to a particular attribute. Therefore, it is difficult to understand how the various attribute-specific analyses interact.

The staff members of the Software Engineering Institute (SEI) documented the relationship between architecture and quality attributes through their work with attribute-based architecture styles (ABASs) [Klein 99a]. However, this effort did not treat the third bullet in the above list.

Notwithstanding the difficulties, systematically codifying the relationship between architecture and quality attributes has several obvious benefits:

- a greatly enhanced architectural design process – for both generation and analysis. During design generation, an architect could reuse existing analyses and determine tradeoffs explicitly rather than on an ad hoc basis. Experienced architects do this intuitively but even they would benefit from codified experience. For example during

analysis, architects could recognize a codified structure and know its impact on quality attributes.

- a method for manually or dynamically reconfiguring architectures to provide specified levels of a quality attribute. Understanding the impact of quality attributes on architectural mechanisms will enable architects to replace one set of mechanisms for another when necessary.
- the potential for third-party certification of components and component frameworks. Once the relationship between architecture and quality attributes is codified, it is possible to construct a testing protocol that will enable third party certification.

For all these reasons, the idea of documenting the relationship between architecture and quality attributes is a worthy goal. This technical note takes another step toward achieving this goal by looking for the quality attribute design primitives that comprise ABASs and architectures.

2 Background

A widely held premise of the software architecture community is that architecture determines quality attributes. This raises several questions:

- Is this premise true and why do we believe that quality attribute behavior and architecture are so intrinsically related?
- Can we pinpoint key architectural decisions that affect specific quality attributes?
- Can we understand how architectural decisions serve as focal points for tradeoffs between several quality attributes?

To address these questions, we will build on the concept of an architectural mechanism introduced by Booch [Booch 96]. By definition, an architectural mechanism is a "structure whereby objects collaborate to provide some behavior that satisfies a requirement of the problem." We are specifically interested in those mechanisms that significantly affect quality attribute behavior and have sufficient content for analysis.

For example, encapsulation is a mechanism for achieving modifiability. Replication is a mechanism for achieving reliability. We contend that these architectural mechanisms allow architects to explore and document the relationship between software architectural styles and quality attributes and perhaps allow for a more methodical use of the ABAS notion.

By codifying mechanisms, architects can identify the choices necessary to achieve quality attribute goals. This, in turn, will set a foundation for further software architecture design and analysis. In fact, this work was motivated by the need for such foundations for the Architecture Tradeoff Analysis Method[SM] (ATAM [SM]) and the Attribute-Driven Design (ADD) method [Kazman 00, Bachmann 00].

Of course, there are many challenges. One challenge is to identify the correct collection of mechanisms at the proper level of abstraction. Another challenge is to understand how to combine mechanisms into styles and styles into systems, while combining the analyses associated with each. We hope that the reader's input will help us to overcome these challenges and reach our goals in a timely manner.

[SM] Architecture Tradeoff Analysis Method and ATAM are servicemarks of Carnegie Mellon University.

3 The Relationship Between Software Architecture and Quality Attributes

In essence, software architectural design, like any other design process, consists of making decisions. These decisions involve

- separating one set of responsibilities from another
- duplicating responsibilities
- allocating responsibilities to processors
- determining how discrete elements of responsibilities cooperate and coordinate

By examining these decisions and their results, we can clarify the relationship between quality attributes and software architecture. Even now, however, we can draw two conclusions:

1. At this level of design, much can be said about many quality attributes.
2. Very little, if anything, must be known about functionality in order to draw quality attribute conclusions.

Let's consider various attributes at a high level. Performance depends on processes, their allocation to processors, communication paths between them, and other factors. Reliability implies redundancy strategies. Modifiability requires dependency chains. In each case, architectural design decisions are necessary to achieve the respective attribute. These decisions require very little knowledge of functionality.

Now let's consider a software architectural decision. Suppose we separate one collection of responsibilities from another. As a result of this decision, we can ask a number of quality attribute questions: What is the impact on modifiability? What are the performance implications? Does this separation have any effect on availability, security, or usability?

All of these questions (couched in a more precise fashion) can be asked and, in some cases, answered as a result of the decision. Thus, every decision embodied in a software architecture can potentially affect quality attributes. At the very least, each decision raises questions about its affect on the attributes. Often these questions cannot be answered by examining the decision, but are answered in the context of additional decisions.

We originally separated the collections of responsibilities for a reason. For example, we might have wanted to support later modifications or to allocate separate portions to different processors to increase performance. We might have wanted to separate computations to improve reliability. Our decision could have been for other reasons. In each case, we can argue how the decision supports some goal. Frequently, that goal is a particular quality

attribute. Thus, whenever we make a software architectural decision that affects quality attributes, there are two consequences:

1. A rationale is formed to explain why the decision helps achieve one (or more) quality attributes.
2. Questions are raised about the impact of the decision on the other attributes—questions that often must be answered in the context of other decisions.

A rationale is important not only because it provides the reasoning for why the decision helps to achieve the attribute, but it will be helpful in understanding the consequences of changing the decision. We call these questions about other attributes "side effects." Before we can analyze the total system for all of the qualities, we must understand these side effects. Applying this to architectural mechanisms, we say that the mechanism achieves some facet of a quality attribute based on a specified reason, while raising questions as side effects of its use.

To summarize, software architecture is closely coupled to how well a system achieves various quality attributes. We can examine portions of the software architecture and either substantiate how each portion achieves a particular quality attribute or generate specific questions it raises about other attributes.

In the next section, we'll look at quality attributes and discuss representing them through the concept of a general scenario.

4 Quality Attributes and General Scenarios

Attribute communities have a natural tendency to expand. This is not surprising because many of the attributes are intertwined. However, at this point, we are not trying to define an attribute by deciding what is within its purview. For example, we do not care whether portability is an aspect of modifiability or whether reliability is an aspect of security. Rather, we want to articulate what it means to achieve an attribute by identifying the yardsticks by which it is measured or observed. To do this, we introduce the concept of a "general scenario." Each general scenario consists of

- the stimuli that requires the architecture to respond
- the type of system elements involved in the response
- the measures used to characterize the architecture's response

For example, a performance general scenario is: "Periodic messages arrive at a system. On the average, the system must respond to the message within a specified time interval." This general scenario describes "events arriving" and being serviced with some "latency." A general scenario for modifiability states "changes arriving" and the "propagation of the change through the system." A security general scenario combines "threats" with a response of "repulsion, detection, or recovery." A usability general scenario ties "the user" together with a response of "the system performs some action."

If an architect or community wants to characterize a particular attribute, a general scenario can be developed to describe it. This addresses the first two problems that we claimed inhibited codifying the relationship between architecture and quality attributes; that is, attribute definitions and their relationships.[1]

General scenarios can apply to any system. For example, we can discuss a change to system platform without any knowledge of the system, because every system has a platform. We also can discuss a change to the representation of a data producer (in a producer/consumer relationship) without knowing anything about the type of data or the functions carried out by the producer. This is not the same as saying that every general scenario applies to every system.

For example, "the user desires to cancel an operation" is one usability general scenario stimulus. This is not the case for every system. For some systems, cancellation is not appropriate. Thus, when applying general scenarios to a particular system, the first step is

[1] Note that even though we advocate defining attributes through general scenarios, attribute names convey meaning and we use them, as appropriate, to define useful categories.

filtering the universe of general scenarios to determine those of relevance. This is essentially the process of determining attribute-specific requirements.

Controlling (or at least significantly affecting) at least one measure of an attribute is the task of an architectural mechanism.

5 Architectural Mechanisms

Earlier we stated that architectural mechanisms directly contribute to quality attributes. Examples of architectural mechanisms are the data router, caching, and fixed priority scheduling. These mechanisms help achieve specific quality attribute goals as defined by the general scenarios.

- The data router protects producers from additions and changes to consumers and vice versa by limiting the knowledge that producers and consumers have of each other. This affects or contributes to modifiability.
- Caching reduces response time by providing a copy of the data close to the function that needs it. This contributes to performance.
- Fixed priority scheduling interleaves multiple tasks to control response time. This contributes to performance.

Each mechanism is targeted at one or more quality attributes. Also, each mechanism implements a more fundamental strategy to achieve the quality attribute. For example, the data router uses indirection to achieve modifiability. Caching uses replication to achieve performance. Fixed priority scheduling uses pre-emptive scheduling to achieve performance.

We can also extend the list of mechanisms and general scenarios to address a wide range of situations. If we introduce a new general scenario, we may need to introduce new mechanisms that contribute to it. Alternatively, we may discover new mechanisms for dealing with existing general scenarios. In either case, we can describe the new mechanism(s) without affecting any other mechanism.

We believe that architectural mechanisms are, in essence, the design primitives for achieving system quality attribute behavior. The goal of our work is to identify the specific design primitives that elicit quality attributes, then analyze those primitives from the point of view of multiple quality attributes.

5.1 Levels of Abstraction for Mechanisms

Mechanisms exist in a hierarchy of abstraction levels. For example, separation is a mechanism that divides two coherent entities. This represents a high level of abstraction. A more concrete mechanism, and lower level of abstraction, would specify entities, such as separating data from function, function from function, or data from data. In this case, each level has meaning and utility with respect to modifiability.

Increasing detail increases the precision of our analysis. However, because our goal is establishing design primitives, we want to determine the highest level of abstraction that still supports some reasoning.

What qualifies as an analysis? Earlier we stated that, by definition, an architectural mechanism helps achieve quality attribute goals. There must be a reason why the mechanism supports those goals. Articulating the reason becomes our analysis. If we cannot state why a particular mechanism supports a quality attribute goal, then it does not qualify as a mechanism. Choosing the correct level of granularity allows us to examine a property at its most fundamental level.

5.2 Desired Attribute and Side Effects

Earlier we also stated that mechanisms have two consequences. They help general scenarios achieve attributes and they can influence other general scenarios as side effects. Given these two consequences, we need two types of analysis. The first analysis will explain how the architectural mechanism helps the intended general scenario achieve its result.

The second analysis will describe how to refine the general scenario in light of the information provided by the mechanism. This analysis reveals side effects that the mechanism has on the other general scenarios. By referring to the specific mechanism that directly affects an attribute, we can perform a more detailed analysis of the side effect.

Let's use encapsulation again as our example. We will analyze the modifiability general scenarios that reflect changes in function, environment, or platform. Our analysis states that as long as changes occur inside the encapsulated function (that is, hidden by an interface), the effect of these changes will be limited. If the expected changes cross the encapsulation boundary, the effect of the changes is not limited.

The analysis/refinement for performance general scenarios seeks to determine whether encapsulation significantly affects resource usage of the encapsulated component.

The analysis/refinement for reliability/availability general scenarios asks whether encapsulation significantly affects the probability of failure of the encapsulated component. A more detailed description of the performance or reliability analyses would be found in appropriate performance or reliability mechanism write-ups.

Table 1 shows the relationship between mechanisms and general scenarios. If mechanisms are across the top, and general scenarios are down the left side, then each cell has one of three possibilities.

1. the mechanism contributes to this general scenario
2. the mechanism has no affect on the general scenario

3. the mechanism introduces side effects—questions to be answered by other mechanisms

Later in this note, we will describe mechanism write-ups for one column of this matrix. That is, each write-up will describe the impact of a mechanism on all of the general scenarios. This doesn't mean that each mechanism will always need a separate section for each general scenario. Rather, we should be able to reason about general scenarios in groups rather than individually.

Table 1. Relationship between Mechanisms and General Scenarios

	Mechanism 1					Mechanism N
General Scenario 1	Analysis	Side effect	Does not apply	Side effect	Analysis	Does not apply
	N/A					
	N/A					
	Side effect					
	Side effect					
	N/A					
	N/A					
General Scenario N	Side effect	N/A	N/A	Analysis		

When describing the analyses to be performed, we group the general scenarios into attribute headings. Thus, in the mechanism template there are sections on modifiability, reliability/ availability, performance, security, and usability. One or more of these is the mechanism's intended attribute. These attributes are presented prior to the attributes that are side effects of the mechanism.

5.3 Mechanisms vis-à-vis ABASs

Attribute-based architectural styles (ABASs) associate classes of architectures with attribute-specific analysis frameworks [Klein 99a, Klein 99b]. Because architectural mechanisms are quality attribute design primitives, we should be able to discuss ABASs using these terms. Consider the following examples:

- The publish/subscribe modifiability ABAS consists of the router and registration mechanisms.
- The tri-modular redundancy (TMR) reliability ABAS consists of the (functional) redundancy and (majority) voting mechanisms.
- The Simplex[SM] reliability ABAS consists of the (analytic) redundancy and (rule-based) voting mechanisms.

[SM] Simplex is a service mark of Carnegie Mellon University

While we intend to explore the relationship between ABASs and mechanisms in the future, we can make several observations right now. Both mechanisms and ABASs are attribute-specific design building blocks containing architectural as well as analysis information.

However, ABASs are "bigger" and more concrete than mechanisms. In the above examples, ABASs comprise more than one mechanism. Yet the type of redundancy and voting mechanisms used by TMR are different from those used by Simplex. Mechanisms, by comparison, are intended to be design primitives and should be indivisible. Moreover, our conjecture is that there are relatively fewer mechanisms than ABASs since there are many ways in which mechanisms can be combined into ABASs. Also, it is our sense from performing architecture evaluations using the ATAM that practicing architects are more familiar with commonly used mechanisms than ABASs. Terms like "redundancy," "voting," and "router" already tend to be in the practitioner's vocabulary.

6 Work Product Templates

So far, we have described two concepts in this note:

1. general scenarios that characterize quality attributes
2. architectural mechanisms

In this section, we present the templates to document attribute stories and mechanisms and indicate the data that each template section should contain.

To understand how architectural mechanisms achieve quality goals, we must first truly understand quality attributes themselves. An attribute story helps us do just that. It creates operational definitions for quality attributes.

6.1 Attribute Story Template

An attribute story presents the issues associated with an attribute and the mechanisms that could help achieve it. We assume that the architect already knows the material in the attribute story. The story simply acts as a checklist for the designer.

In Section 7, we will present attribute stories for modifiability and performance. Attribute stories for reliability/availability, security, and usability will appear in subsequent versions of this note.

Concepts of <the attribute>

This section describes the attribute including relevant stimuli and typical response measures.

General <attribute> scenarios

This section presents a number of system-independent or general scenarios. General scenarios describe the attributes. These high-level scenarios could apply to any system. Depending on the general scenario, the system can be characterized by mechanisms, modules, or descriptions of the hardware platform.

Strategies for achieving <attribute>

This section describes the basic strategies used to achieve the general scenarios. This sets the foundation for the next section of the template.

Enumeration of mechanisms for achieving <attribute>

This section lists the names and describes the mechanisms used to achieve the attribute. Each mechanism will have its own description. The brief descriptions here serve as a checklist for architects.

6.2 Mechanism Template

A mechanism template provides information the architect needs to reason about a specific mechanism for the five attributes under discussion.

Description of this mechanism

This section describes a particular mechanism.

Analysis of this mechanism

This section describes the strategy that the mechanism used to achieve attribute goals. It also presents an argument that the mechanism does, in fact, support one or more general scenarios. The following sections apply to the attributes whose properties exist as side effects to the mechanism.

Modifiability general scenarios

This section interprets modifiability general scenarios through the prism of the mechanism.

Performance general scenarios

This section interprets performance general scenarios through the prism of the mechanism.

Reliability/Availability general scenarios

This section interprets reliability/availability general scenarios through the prism of the mechanism.

Security general scenarios

This section interprets security general scenarios through the prism of the mechanism.

Usability general scenarios

This section interprets usability general scenarios through the prism of the mechanism.

7 Attribute Stories

In this section, we present attribute stories for two of the five attributes under discussion.

7.1 The Modifiability Story

Concepts of modifiability

Modifiability is the ability of a system to be changed after it has been deployed. In this case, the stimulus is the arrival of a change request and the response is the time necessary to implement the change. Requests can reflect changes in functions, platform, or operating environment. A request could also modify how the system achieves its quality attributes. These changes can involve the source code or data files. They can be made at compile time or run time.

General modifiability scenarios[1]

A request arrives to change the functionality of the system. The change can be to add new functionality, to modify existing functionality, or to delete a particular functionality.

- The hardware platform is changed. The system must be modified to continue to provide current functionality. There may be a change in hardware including input and output hardware, operating system, or COTS middleware.
- The operating environment is changed. For example, the system now has to work with systems previously not considered. It may have to operate in a disconnected mode. It may have to dynamically discover what devices are available, or it may have to react to changes in the number or characteristics of users.
- A request arrives to improve a particular quality attribute such as reliability, performance, or usability. The system should be modified to achieve better usability, for instance.
- A new distributed user arrives and wishes to utilize just some of the services available.
- As a result of quality considerations, some portion of the system modifies its behavior during run time.

The general form for modifiability scenarios is

Time of modification ::= Compile time | run time

Type of modification ::= Add | modify | delete

[1] Note that while our ultimate goal is to be relatively comprehensive, we make no claim of being close to comprehensive in this paper.

Target of modification ::= Functionality | platform (hardware, OS, middleware) | environment (interoperate, disconnect, #users), quality attribute

Strategies that achieve modifiability

At compile time, modifiability is achieved through three basic strategies.

1. Indirection – Indirection involves inserting a mediator, such as a data router, to allow variation within a particular area of concern. For example, when producers and consumers of data communicate via a mediator, they typically have no direct knowledge of each other. This means that additional producers or consumers could be added without change to the consumers or producers. A virtual device or machine can also serve as a mediator in the indirection strategy.
2. Separation – This strategy separates data and function that address different concerns. Since the concerns are separate, we can modify one concern independently of another. Isolating common function is another example of a separation strategy.
3. Encoding function into data meta-data and language interpreters – By encoding some function into data and providing a mechanism for interpreting that data, we can simplify modifications that affect the parameters of that data.

At run time, modifiability is achieved using a strategy of dynamic discovery and reconfiguration. This strategy involves discovering the resources or functions that become dynamically available as the system's environment changes, negotiating for access to those resources, then accessing the resource.

Mechanisms that achieve modifiability

- data router (Also called a data bus or a data distributor) – This intermediary mechanism directs (and possibly transforms) data from producers to consumers. It requires a supporting mechanism to identify producers and consumers.
- data repository – This mechanism stores data for later use.
- virtual machine – This mechanism places an intermediary between function users and producer(s).
- independence of interface from implementation – This mechanism allows architects to substitute different implementations for the same functionality. Interface definition languages (IDLs) are an example of this mechanism.
- interpreter – This mechanism involves encoding function into parameters and more abstract descriptions to allows architects to easily modify them.
- client-server – This mechanism involves providing a collection of services from a central process and allowing other processes to use these services through a fixed protocol.

7.2 The Performance Story

Concepts of performance

Performance involves adjusting and allocating resources to meet system timing requirements. The stimulus can be any event that requires a response, such as the arrival of a message, the expiration of an interval of time (as signaled by a timer interrupt), the detection of a significant change in the system's environment, a user input event, etc.

In this situation, the system consists of a collection of processors connected by a network, involving "schedulable entities" such as processes, threads, and messages. Important response measures include

- latency – the amount of time that passes from the event occurrence until the completed response. Average-case latency matters and worst-case latency are variants of latency measures
- throughput – the number of events responded to per unit of time
- precedence – preserving ordering constraints on the responses to events
- jitter – variance in the time interval between response completions
- growth capacity – ability to add load while meeting timing requirements

At a higher level, performance concerns the ability to predictably provide various levels of service (measured in the above terms) under varying circumstances. Performance depends upon the load to the system and the resources available to process the load.

General performance scenarios

The general form of general performance scenarios is

<Events arrive periodically, sporadically **or** randomly> and must be completed <in the worst-case **or** on the average> <within a time interval **and/or** subjected to data loss constraints **and/or** subjected to throughput constraints **and/or** constrained by order relative to other event responses **and/or** adhering to jitter constraints>

A sample of general performance scenarios assuming unchanging resource requirements follows:

- Periodic or sporadic events (that is, events with bounded inter-arrival times) arrive. In all cases, the system must execute its response to the event within a specified time interval. (One common example is the time interval that starts when the event occurs and ends one period later.)
- Periodic or sporadic events arrive. In all cases, no more than n/m events can be lost.
- Periodic or sporadic events arrive. On average the system must execute its response to the event within a specified time interval.

- Periodic or sporadic events arrive. On average no more than n/m events can be lost.
- Events arrive stochastically. On average the system must execute its response to the event within a specified time interval. The response, however, should never be greater than a specified upper bound.
- Events arrive stochastically. On average the system must complete n responses per unit time.

We can extend the above general performance scenario template to include cases where the resource requirements change. These cases include

- Periodic or sporadic events arrive.
- In all cases, the system must carry out its response to the event within a specified time interval by executing n nodes in a network, where each node consists of m processors on a bus and each processor is executing p processes. If a subset of the resources fails, the most critical subset of event responses must continue to be met.

The most general form for a performance scenario considers many or all of the following aspects:

- arrival pattern ::= periodic | sporadic | random
- latency requirement ::= worst-case | average-case [with upper bound] | window
- throughput requirement ::= n completions per unit time
- data loss requirement ::= no more than m of n can be lost
- platform ::= uniprocessor | distributed
- resources ::= processors, networks, buses, memory
- mode ::= (resource driven | load driven), (discrete | continuous)

Strategies that achieve performance

Performance is achieved by managing resources (such as CPUs, buses, networks, memory, data, etc.) that affect system response. That is, timing behavior is determined by allocating resources to resource demands, choosing between conflicting requests for resources, and managing resource usage. Therefore, there are three key techniques for managing performance:

1. resource allocation – These are policies for realizing resource demands by allocating various resources; for example, deciding which of several available processors and process will execute.
2. resource arbitration – These are policies for choosing between various requests of a single resource; for example, deciding which of several processes will execute on a processor. Arbitration includes preemptive strategies and exclusive use strategies.
3. resource usage – These are strategies for affecting resource demands; for example, deciding how to reduce the execution time needed to respond to one or more events, or

deciding not to execute certain tasks in periods of intense computation of high-priority tasks.

Each of these aspects affects the completion time of the response. As a result, each aspect can be viewed as a category of mechanism.

Mechanisms that achieve performance[1]

In general, performance can be achieved through resource allocation, resource arbitration, and resource use.

Resource allocation

- load balancing – spreading the load evenly between a set of resources
- bin packing – based on measures of spare capacity (for example, schedulable utilization) Bin packing maximizes the amount of work that can be performed by a collection of resources.

Resource arbitration

Preemptive

- dynamic priority scheduling – allows the priority of a process to vary from one invocation to the next (e.g., earliest deadline first scheduling)
- fixed priority scheduling – the priority of processes are fixed from one invocation to the next (e.g., rate monotonic analysis or deadline monotonic)
- aperiodic service strategies – mechanisms for scheduling aperiodic processes in a periodic context (e.g., slack stealing)
- static scheduling – timelines which are calculated prior to runtime (e.g., cyclic executive)
- queue overwrite policy – mechanism for handling overflow in a bounded queue (e.g., overwrite oldest)
- QoS-based methods – methods that explicitly consider quality of service in scheduling decisions (e.g., QRAM)
- overload management/load shedding – mechanisms that determine which processes receive resources when the system cannot service all requests

Exclusive use

- synchronization policies – mechanisms for managing mutually exclusive access to shared resources (e.g., priority inheritance)

Resource use

- caching – using a local copy of data to reduce access time

[1] Note that while our ultimate goal is to be relatively comprehensive, we make no claim of being close to comprehensive in this paper.

8 Mechanisms

In this section, we present two sample mechanisms.

8.1 Data Router

Description of this mechanism

The data router (also called a data bus or a data distributor) mechanism implements the strategy of data indirection. It inserts a mediator between a data item producer and consumer. By providing a mediator, the data producer has embedded knowledge that it must send the data item to the mediator. The mechanism that the data router uses to recognize consumers for a data item is discussed separately (see data registration). Figure 1 describes this mechanism.

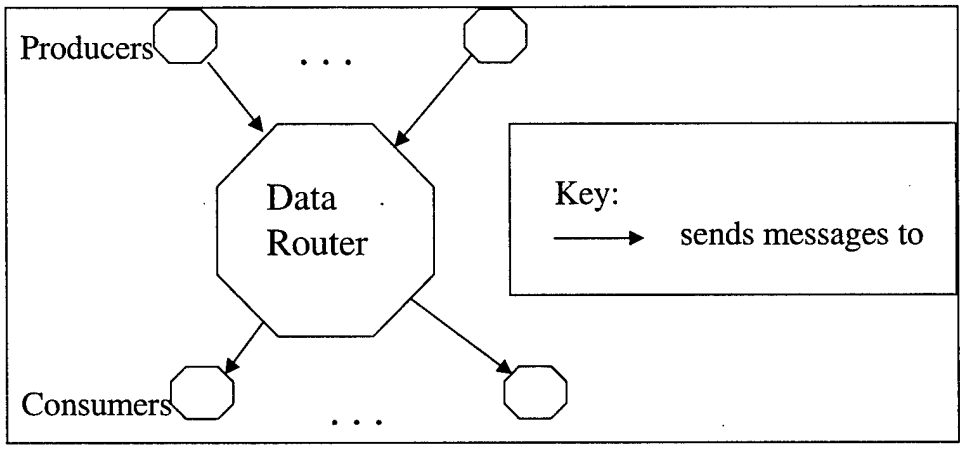


Figure 1. Data Router

Producers are components that generate messages. Consumers are components that receive messages. It is possible for a component to both produce and consume different types of messages. The router directs messages of particular type to consumers of that type. We do not show the registration mechanism that the data router uses to identify producers and consumers.

Sometimes the data router also converts data. For example, suppose the data producer of a speed measurement supplies it in kilometers per hour and the consumer wishes to receive it in miles per hour. To perform the conversion, the data router must know the units in which the data item is produced and the units that the consumer desires.

Purpose of this mechanism

The data router mechanism reduces the coupling between the data producer and consumer. This simplifies the process of adding new consumers or producers of a single type of data.

The stimulus in the following general scenarios changes function, platform, or environment:

- adding/removing/modifying a producer/consumer of a data item within a known data type. Adding or removing a producer/consumer of a data item should not affect the producer, since the producers and consumers do know each other. The new consumer or producer must be registered to the data router. (Registration is outside the scope of this mechanism.) If the producer being removed is the only element that supplies that data type, then consumers will not receive any data. Presumably, that is the intent of the modification.
- adding a new data type to a producer of a data item. Adding a new producer data type should not affect any existing consumers since they can remain ignorant of the new data and its producers. The mediator may need to be modified if it has to transform the new data type. None of the existing producers are also affected by adding a new producer.
- adding a new conversion type to the data router. The data router needs to be modified to perform the new conversion. Existing conversions and producers and consumers are not modified.
- modifying the data router. The effect of modifying the data router depends, obviously, on the type of modification. Adding a new data type should not affect existing data types. Modifying the interfaces of the data router will require all producers and consumers to be modified.

Performance general scenarios

- What is the mean/largest/minimum size (bytes)/distribution of data items?
- Can multiple messages arrive at the mediator simultaneously?
- What is the maximum/mean number of consumers for each data item?
- What is the delay introduced by the mediator for the consumer of a data item?
- Are messages produced periodically (at what period) or episodically (at what rate and distribution)?
- Are the producers and consumers in the same process or in different processes?
- If the producers and consumers are in different processes (or on different processors) what is the latency of message delivery? What are the resources involved in message delivery?
- How are producers and consumers synchronized?

Security general scenarios

- Can eavesdroppers listen to the transmission of any data items to/from the data router?
- Is it possible for unauthorized producers to send messages to the mediator or for unauthorized consumers to receive messages from the mediator?

Availability/reliability general scenarios

- What is the likelihood/impact if a produced message does not arrive?
- What is the likelihood/impact of a message not being produced in time?

Usability general scenarios

- What data items sent to the mediator should be visible to the user?
- What feedback should the user receive about the messages sent to/from the mediator?
- What needs to be recorded in order to cancel, undo, or to evaluate the system for usability?

8.2 Fixed Priority Scheduling

Description of this mechanism

Process scheduling is a common mechanism provided by most operating systems. It automatically interleaves execution of units of concurrency (such as processes and/or threads). Fixed priority scheduling is a uni-processor scheduling policy that prioritizes each process/thread in the system. The scheduler uses this priority to decide which process to execute. It assigns the processor to the highest priority process/thread that is not currently blocked.

Figure 2 presents a module view of a fixed priority scheduler and Figure 3 presents a sequence diagram of a fixed priority scheduler with two tasks. The high-priority task executes until it blocks. The next priority task then executes until the high-priority task is ready to execute again.

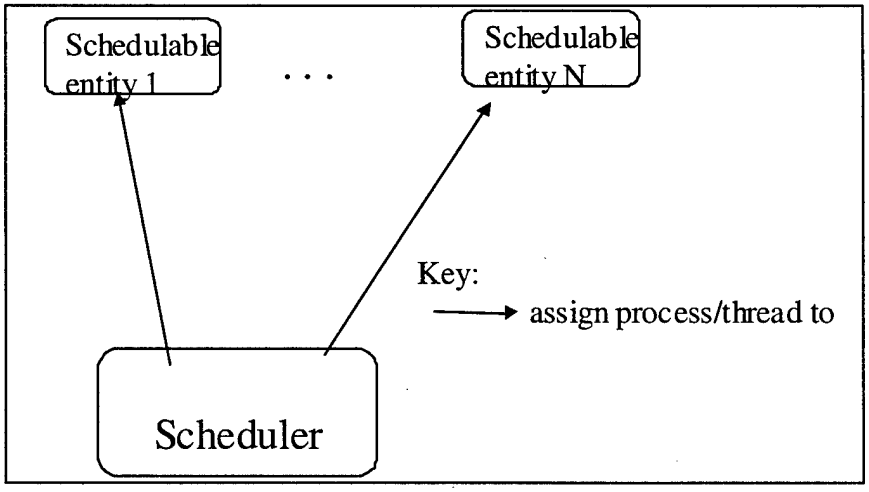


Figure 2. Fixed Priority Scheduler

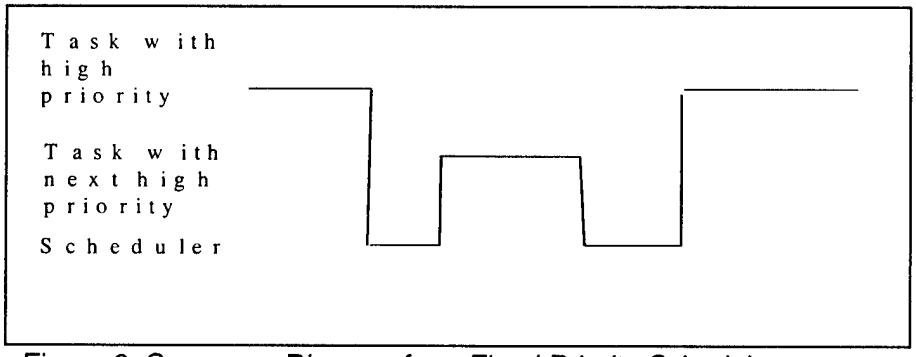


Figure 3. Sequence Diagram for a Fixed Priority Scheduler

Purpose of this mechanism

The purpose of any scheduling mechanism is two-fold:

1. separate the management of concurrency and the concomitant performance (i.e., timing) concerns from the concerns of implementing of functionality. This allows the architect to design a structure that reflects the "natural concurrency" of the system's environment (such as handling multiple message types, or sensor types that have disparate timing properties).
2. allocate a single processor to event stimuli in the manner required by their timing requirements

Fixed priority scheduling allows for a level of predictability that corresponds to the timing estimates associated with the various processing scenarios. This predictability is due to scheduling analysis techniques (such as rate monotonic analysis).

The set of general scenarios addressed by this mechanism are:

<Events arrive periodically **or** sporadically> and must be completed <in the worst-case > <within a time interval **and/or** subjected to data loss constraints> <while executing on a uniprocessor>.

We will make two assumptions for this analysis:

1. The dominant character of processes are deterministic.
2. The only effects of concern for this mechanism derive from allocating the processor to processes. (The effects of processes sharing data, acquiring memory, or using hardware devices are not considered here.)

Simplest case:
In this simple case, the processes are the only sources of execution time, the processes are preemptable by higher priority processes, and no pipelines exist in the topology.

As you can see, only two factors affect the latency of a process, (1) its own execution time and (2) preemption due to higher priority processes. Latency can be computed by determining the fixed point of the following expression where,

C_i is the execution time of process

i, the process whose latency is being computed, and

the term with the summation computes the effects of preemption:

$$l = \sum_{j \in H} \left\lceil \frac{l}{T_j} \right\rceil C_j + C_i$$

Other factors:
How are pipelines analyzed?
A pipeline is an ordered sequence of schedulable entities. That is, the *nth* element in the sequence is blocked until the n-1st element has been completely executed. The key to analyzing a pipeline (assume it is the ith pipeline) is to identify the lowest priority process (LowPi) in the pipeline. Other pipelines are then placed into sets depending whether they have processes whose priorities are greater than or equal to LowPi and whether the higher priority processes are at the beginning of the pipeline or not. Each pipeline in which all processes have priorities greater than LowPi can preempt more than once. (Call this the H set.) Each pipeline that starts at a higher priority and then decreases to a lower priority can preempt once. (Call this the HL set) Of all of the pipelines that start at a lower priority than LowPi and then increase to a higher priority than LowPi, only one can have a delaying effect. (Call this the LH set.)

How are non-preemptable sections accounted for?
Add a term in the above equation which is equal to the execution time of the longest non-preemptable section.

How are interrupts accounted for?
Depending on when the interrupt is reenabled, treat it either as a high-priority process (reenabled at the end of the interrupt service routine) or as an HL pipeline (reenabled after some lower priority execution takes place).

How is operating system overhead accounted for?
This depends on whether the overhead is directly associated with the scheduling of a process, whether it is associated with handling some other OS function, and whether the overhead occurs at a higher priority (for example, in a non-preemptable section).

If the overhead occurs as part of dispatching a process, this extra execution time can be added to the execution of the process. If, in this case, the overhead executes at a higher priority, it can be treated like a new process in a pipeline.

If the overhead occurs semi-regularly and is totally independent of scheduling a specific process, then treat it as an additional periodic process.

What if the operating system doesn't support a sufficient number of priority levels?
When a process is assigned the same priority as another process (which would have been assigned a lower priority if there were sufficient priority levels), assume that this now-equal priority process can preempt it.

Performance considerations

The following questions are specifically relevant to fixed priority scheduling:

- What is the execution time of processes? Is the maximum known?
- What are other sources (other than the processes) of execution time (such as context switch, dispatching, or daemons for garbage collection)?
- How many priority levels does the scheduler support?
- What are the sources of interrupts or non-preemptable sections?
- How often do the processes execute? Is there a minimum amount of time between executions?
- What is the flow of control/data between processes?
- Are there various modes in which the above characteristics change?
- What is the stochastic character of processing frequency and execution time?

Modifiability considerations

In general, any of the performance parameters mentioned in the preceding section might change. Therefore, the following questions could be asked:

- What is the impact of adding or deleting new processes?
- What is the impact of changing process priorities?
- What is the impact of changing process execution times?
- What is the impact of changing event arrival rates?

- What is the impact of switching operating systems?

Reliability considerations

- How can errors propagate between processes?
- Do processes run in their own address space?
- What happens if processes exceed their so-called worst-case execution estimates?

Security considerations

- Can processes be killed and then spoofed?
- Can new processes be created (with the intent of causing denial of service)?

Usability considerations

- How is average response time managed?
- Do any of the processes generate data from which the user gets feedback?
- Is consideration of this feedback data included in the setting of deadlines?

9 Future Work

Over the next several months, we hope to complete the attribute stories and mechanism write-ups. In addition, we plan to continue codifying the relationship between architecture and quality attributes. Specifically, we want to address the following tasks:

- Understand the relationship between mechanisms and styles. ABASs are architectural styles that have associated analysis frameworks. If, as we believe, an architectural style is a combination of several mechanisms, there should be a relationship between the mechanism analyses and the ABAS analysis.
- Refine and further develop a collection of ABASs.
- Describe "vertical domains" (similar to Buschmann's systems of patterns) using mechanisms and ABASs.
- Embed the codification into a database. The codification represents a collection of knowledge. It should support various off line searching strategies so that architects could weigh their options during the design process.
- Encode the codification into a rule base that supports dynamic on-line reconfiguration using multi-attribute quality of service criteria.
- Integrate ABASs and mechanisms into the Attribute-Driven Design (ADD) method.
- Better integrate ABASs and mechanisms into the ATAM.

10 References

[Bachman 00] Bachmann, F.; Bass, L.; Chastek, G.; Donohoe, P. & Peruzzi, F. *The Architecture Based Design Method* (CMU/SEI-2000-TR-001 ADA375851). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. Available WWW: <URL: http://www.sei.cmu.edu/publications/documents/00.reports/00tr001.html>.

[Bass 98] Bass, L.; Clements, P. & Kazman, R. *Software Architecture in Practice.* Reading, MA: Addison Wesley Longman, 1998.

[Bergey 01] Bergey, J.; Barbacci, M. & Wood, W. *Using Quality Attribute Workshops to Evaluate Architectural Design Approaches in a Major System Acquisition: A Case Study* (CMU/SEI-2000-TN-010). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001. Available WWW: <URL: http://www.sei.cmu.edu/publications/documents/00.reports/00tn010.html>.

[Boehm 78] Boehm, B. et al. *Characteristics of Software Quality.* New York: Elsevier North-Holland Publishing Company, Inc., 1978.

[Booch 96] Booch, G. *Object Solutions: Managing the Object-Oriented Project.* Reading, MA: Addison Wesley Longman, 1996.

[Gamma 95] Gamma, E.; Helm, R.; Johnson, R. & Vlissides, J. *Design Patterns.* Reading, MA: Addison Wesley Longman, 1995.

[ISO 91] International Organization for Standardization (ISO), "Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for Their Use," ISO/IEC 9126, 1996

[Kazman 00] Kazman, R.; Klein, M. & Clements, P. *ATAM: Method for Architecture Evaluation* CMU/SEI-2000-TR-004 ADA382629. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. Available WWW: <URL: http://www.sei.cmu.edu/publications/documents/00.reports/00tr004.html>.

[Klein 99a] Klein, M.; Kazman, R.; Bass, L.; Carriere, S.J.; Barbacci, M. & Lipson, H. "Attribute-Based Architectural Styles," 225-243. *Proceedings of the First Working IFIP Conference on Software Architecture* (WICSA1). San Antonio,

TX: February 1999.

[Klein 99b] Klein, M. & Kazman, R. *Attribute-Based Architectural Styles* (CMU/SEI-99-TR-022 ADA371802). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1999. Available WWW: <URL: http://www.sei.cmu.edu/publications/documents/99.reports/99tr022/99tr022.html>.

[Kleinrock 76] Kleinrock, L; *Queuing Systems*; NY: Wiley, 1976.

[Lyu 96] Lyu, M. R., editor in chief; *Handbook of Software Reliability Engineering*, New York: McGraw Hill; Los Alamitos, Calif.: IEEE Computer Society Press, c1996.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	December 2000	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Quality Attribute Design Primitives	F19628-00-C-0003

6. AUTHOR(S)

Len Bass, Mark Klein, Felix Bachmann

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2000-TN-017

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

This report focuses on the quality attribute aspects of mechanisms. An architectural mechanism is a "structure whereby objects collaborate to provide some behavior that satisfies a requirement of the problem." The authors identity mechanisms that significantly affect quality attribute behavior and have sufficient content for analysis. Codifying such mechanisms will enable architects to identify the choices necessary to achieve quality attribute goals.

14. SUBJECT TERMS	15. NUMBER OF PAGES
software architecture, quality attributes, software mechanisms, Attribute-based architectural styles, ABAS,	42

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18 298-102